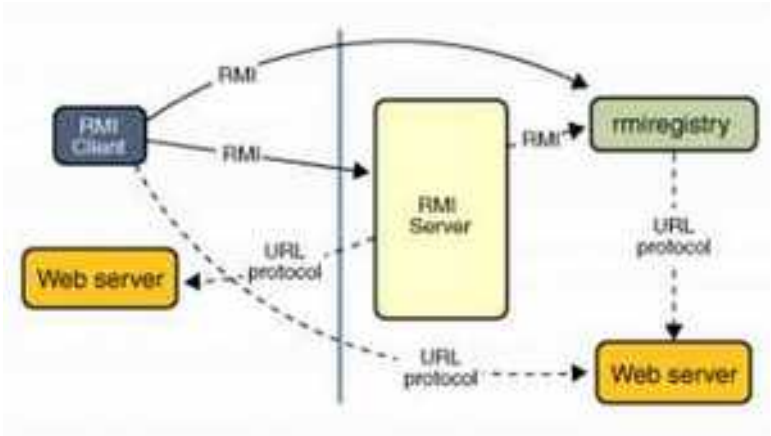
Лабораторная работа №5. Клиент. Сервер. RMI

Теоретическая часть

Одной из интереснейших технологий платформы Java является удалённый вызов методов (Remote Method Invocation, RMI). Эта технология позволяет строить распределённые приложения, поскольку с её помощью Java-объекты, выполняющиеся на одном компьютере, могут вызывать методы Java-объектов, которые выполняются на другом компьютере.

Рассмотрим основные принципы применения RMI. Разработаем распределенное приложение, называемое вычислителем. Вычислитель, удаленный объект на сервере, получает задачу от клиентов, выполняет ее и возвращает какие-то результаты. Задачи выполняются на той машине, на которой выполняется сервер. Новым аспектом вычислителя является то, что задачи, выполняемые им не нужно определять, когда вычислитель пишется. Новый вид задач может быть создан в любой момент и затем придан для выполнения на вычислителе. Все, что требуется от задачи, это то, чтобы ее класс реализовал определенный интерфейс. Такая задача может быть послана на вычислитель и запущена, даже если класс, который определяет эту задачу, будет написан после того как был написан и запущен вычислитель.

Код, необходимый для выполнения задачи, может быть "скачан" системой RMI для вычислителя, а затем вычислитель выполняет задачу, используя ресурсы машины, на которой выполняется вычислитель. RMI динамически загружает код задачи в виртуальную машину Java вычислителя и выполняет задачу без априорного знания класса, который реализует задачу.



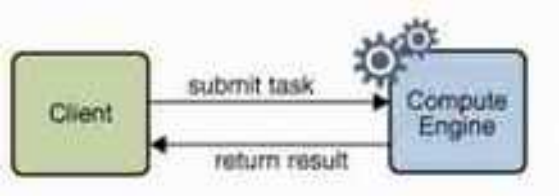
Написание сервера RMI

Сервер вычислителя принимает задачи от клиентов, выполняет задачи и возвращает любые результаты. Сервер включает в себя интерфейс и класс. Интерфейс обеспечивает определение методов, которые могут быть вызваны клиентом. В сущности, интерфейс определяет представление клиента об удаленном объекте. Класс обеспечивает реализацию.

Проектирование удаленного интерфейса

Ядром вычислителя является протокол, который:
1) позволяет работам быть посланными на вычислитель,
2) вычислителю запускать эти работы
3) и результатам работы возвращаться к клиенту.

Этот протокол выражается в интерфейсах, поддерживаемых вычислителем и в объектах, которые посылаются на вычислитель, и показан на следующем рисунке.



Каждый из интерфейсов состоит из единственного метода. Интерфейс вычислителя, Compute, позволяет работам быть посланными на вычислитель, клиентский интерфейс, Task, определяет, как вычислитель выполняет полученную задачу. Вот удаленный интерфейс с его единственным методом:

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
  <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Расширением интерфейса java.rmi.Remote, интерфейс Compute помечает себя как одного из тех, для кого его методы могут быть вызваны из любой виртуальной машины. Любой объект, который реализует этот интерфейс становится удаленным объектом. Как член удаленного интерфейса, метод executeTask является удаленным методом. Следовательно, метод должен быть определен с возможностью выбрасывания java.rmi.RemoteException. Это исключение выбрасывается системой RMI во время вызова удаленного метода для того, чтобы показать, что произошел сбой связи или ошибка протокола. RemoteException - это проверочное исключение, так что не требуется никакого кода в удаленном методе для обработки этого исключения, кроме вылавливания его или объявления его в выражении throws.

Второй интерфейс, необходимый для вычислителя, определяет тип Task. Этот тип используется как аргумент метода executeTask в интерфейсе Compute. Интерфейс compute.Task определяет интерфейс между вычислителем и работой, которую он должен выполнить, обеспечивая способ запуска работы.

```
package compute;

public interface Task<T> {
  T execute();
}
```

Описание интерфейса Task является типизированным с параметром T.

RMI использует механизм сериализации объектов Java для их передачи (передача параметра метода по значению) между виртуальными машинами java.

Классы, реализующие интерфейс Task могут содержать любые данные, необходимые для расчета и любые методы.

Вычислитель, реализованный классом ComputeEngine, реализует интерфейс Compute, позволяя передавать ему различные задачи вызовом его метода executeTask. Эти задачи выполняются, используя реализацию задачи в методе execute.

Реализация удаленного интерфейса

В общем случае класс удаленного интерфейса должен, как минимум:

1) объявить удаленный интерфейс, который реализуется;
2) определить конструктор удаленного объекта;
3) обеспечить реализацию для каждого метода удаленного интерфейса.

Серверу необходимо создавать и инсталлировать удаленные объекты. Эта процедура установки может быть сокрыта в методе main в самом классе, реализующем удаленный объект, или может быть включена в совершенно другой класс.

Процедура установки должна:

1) создать и инсталлировать менеджер безопасности;
2) создать один или более экземпляров удаленного объекта;
3) зарегистрировать как минимум один из удаленных объектов в регистре RMI (или в другом сервисе именования, таком как JNDI[1]), для самозагрузки.

Полная реализация вычислителя приводится ниже. Класс engine.ComputeEngine реализует удаленный интерфейс Compute и также включает метод main для установки вычислителя.

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
```

1Java Naming and Directory Interface (JNDI) — это набор Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам.

```
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
          Registry registry = LocateRegistry.getRegistry();
          registry.rebind(name, stub);
          System.out.println("ComputeEngine bound");
        } catch (Exception e) {
          System.err.println("ComputeEngine exception:");
          e.printStackTrace();
        }
      }
    }
```

Теперь давайте поближе рассмотрим каждый из компонентов реализации вычислителя.

UnicastRemoteObject - удобный класс, определенный в RMI API, который может использоваться как суперкласс для реализаций удаленного объекта. Суперкласс UnicastRemoteObject предоставляет такую реализацию многих методов класса java.lang.Object (equals, hashCode, toString), что они определяются подходящим образом для удаленных объектов. UnicastRemoteObject также включает в себя конструкторы и статические методы, используемые для экспорта удаленных объектов, который делает удаленный объект доступным для получения по запросам от клиентов.

Реализация удаленного объекта не обязана быть расширением UnicastRemoteObject, но любая реализация, которая этого делает, должна предоставлять соответствующую реализацию методов java.lang.Object. Кроме того, реализация удаленного объекта должна делать явный вызов методов exportObject класса UnicastRemoteObject, чтобы уведомить среду выполнения RMI об удаленном объекта, чтобы объект мог принимать приходящие вызовы.

Класс ComputeEngine также содержит некоторые методы, которые могут вызываться только локально:

1) конструктор для объектов ComputeEngine;

2) метод main, который используется для того, чтобы создать ComputeEngine и сделать его доступным для клиентов.

Класс ComputeEngine имеет единственный конструктор, который не имеет аргументов. Код конструктора:

```
public ComputeEngine() {
  super();
}
```

Этот конструктор просто вызывает конструктор суперкласса, который является конструктором без аргументов класса UnicastRemoteObject. Хотя конструктор суперкласса вызовется даже если такой вызов будет опущен в конструкторе ComputeEngine, мы включили его для ясности.

В ходе конструирования UnicastRemoteObject становится экспортированным, это означает, что он доступен для поступающих запросов путем прослушивания поступающих от клиентов запросов на анонимном порту.

Конструктор без аргументов суперкласса UnicastRemoteObject объявляет исключения RemoteException в своем выражении throws, так что конструктор ComputeEngine также объявляет, что он может выбрасывать RemoteException. RemoteException может происходить во время конструирования, если попытка экспортирования объекта собьётся - из-за того, например, что недоступны коммуникационные ресурсы или не найден соответствующий класс заглушки.

Класс удаленного объекта обеспечивает реализацию для каждого удаленного метода, определенного в уделенном интерфейсе. Интерфейс Compute содержит единственный метод, executeTask, который реализован так:

```
public <T> T executeTask(Task<T> t) {
  return t.execute();
}
```

Этот метод реализует протокол между ComputeEngine и его клиентами. Клиент обеспечивает ComputeEngine объектом Task, который имеет реализацию метода задачи execute. ComputeEngine выполняет Task и возвращает результат выполнения метода задачи execute прямо вызывающему объекту.

Методу executeTask нет необходимости знать что-нибудь еще о результате метода execute, кроме того, что это он возвращает Task<T>. Вызывающий объект, повидимому, знает больше о типе возвращаемого Task<T> и может преобразовать результат в соответствующий тип.

Передача объектов в RMI

Передаваемые аргументы или возвращаемые значения методов могут быть почти любого типа, включая локальные объекты, удаленные объекты и примитивные типы. Точнее, любая сущность любого типа может быть передана в или из удаленного метода, если эта сущность является экземпляром типа, который является примитивным типом, удаленным объектом или сериализуемым объектом, что значит, что он должен реализовать интерфейс java.io.Serializable.

Несколько типов объектов не соответствуют этим критериям и не могут быть переданы или возвращены удаленным методом. Большинство этих объектов, такие как дескриптор файла, инкапсулируют информацию, которая имеет смысл только в единственном адресном пространстве. Многие из базовых классов, включая классы пакетов java.lang и java.util, реализуют интерфейс Serializable.

Правила, управляющие тем, как передаются аргументы и возвращаемые значения, следующие:

1) Удаленные объекты, в сущности, передаются по ссылкам. Ссылка удаленного объекта - это заглушка, являющаяся его представителем на клиентской стороне, которая реализует полный набор удаленных интерфейсов, реализуемых удаленным объектом.

2) Локальные объекты передаются копией, с использованием сериализации объектов. По умолчанию копируются все поля, кроме тех, которые отмечены static или transient[2]. Поведение по умолчанию при сериализации может быть замещено. Передача объектов по ссылке (как это делается с удаленными объектами) означает, что любые изменения, сделанные в состоянии объекта удаленным методом, отражаются в оригинале удаленного объекта. Когда передается удаленный объект, для получателя доступны только те интерфейсы, которые являются удаленными интерфейсами; любые методы, определенные в реализующем классе или в неудаленных интерфейсах, реализуемых классом, недоступны для получателя.

Например, если вы передаете ссылку на экземпляр класса ComputeEngine, получатель будет иметь доступ только к методу вычислителя executeTask. Этот получатель

2Сериализация - это преобразование экземпляра класса в форму, пригодную для его сохранения (например в файл, в БД или дя передачи по сети). Сериализованные объекты можно затем восстановить (десериализовать). Свойства класса, помеченные модификатором transient, не сериализуются. Обычно в таких полях хранится промежуточное состояние объекта, которое, к примеру, проще вычислить, чем сериализовать, а затем десериализавать. Другой пример такого поля - ссылка на экземпляр объекта, который не требует сериализации или не может быть сериализован.

даже не будет видеть конструктор ComputeEngine или его метод main или любые методы java.lang.Object.

В вызове удаленного метода объекты - параметры, возвращаемые значения и исключения - которые не являются удаленными объектами, передаются по значению. Это значит, что создается копия объекта на получающей виртуальной машине. Любые изменения в состоянии этих объектов у получателя отражаются только на копии получателя, а не в оригинальном экземпляре.

Реализация метода main сервера

Наиболее сложным методом в реализации ComputeEngine является метод main. Метод main используется для запуска ComputeEngine и, следовательно, должен сделать необходимые инициализации и действия для подготовки сервера к принятию запросов от клиентов. Этот метод не является удаленным методом, что означает, что он не может быть вызван из другой виртуальной машины. Поскольку метод main объявлен static, метод вообще не связан с объектами, кроме класса ComputeEngine.

1. Создание и инсталляция менеджера безопасности

Первое, что делает метод main, это создание и инсталляция менеджера безопасности, который защищает системные ресурсы от выполнения на виртуальной машине непроверенных "скачанных" кодов. Менеджер безопасности определяет, имеют ли "скачанные" коды доступ к локальной файловой системе и могут ли они выполнять другие привилегированные операции.

Все программы, использующие RMI, должны устанавливать менеджер безопасности, иначе RMI не "скачает" классы (кроме как по локальному classpath) для объектов, получаемых как параметры, возвращаемые значения или исключения в вызовах удаленных методов. Это ограничение гарантирует, что операции, выполняемые "скачанным" кодом проходят через ряд проверок безопасности.

ComputeEngine использует менеджер безопасности. Вот код, который создает и инсталлирует менеджер безопасности:

```
if (System.getSecurityManager() == null) {
   System.setSecurityManager(new SecurityManager());
}
```

2. Обеспечение доступности удаленного объекта для клиентов

Далее метод main создает экземпляр ComputeEngine. Это делается оператором

```
Compute engine = new ComputeEngine();
```

Как было сказано, этот конструктор вызывает конструктор суперкласса UnicastRemoteObject, который экспортирует вновь созданный объект в среду выполнения RMI. Когда шаг экспорта завершен, удаленный объект ComputeEngine готов принимать поступающие вызовы от клиентов на анонимном порте, который выбран RMI или низкоуровневыми средствами операционной системы. Заметьте, что типом переменной engine является Compute, а не ComputeEngine. Это объявление подчеркивает, что интерфейс, доступный для клиента - это интерфейс Compute с его методами, а не класс ComputeEngine с его методами.

3. RMI

Прежде чем вызывающий объект сможет вызывать методы удаленного объекта, этот вызывающий объект должен получить ссылку на удаленный объект. Это может быть сделано так же, как программой получаются другие объектные ссылки, получением ссылки как части возвращаемого значения метода или как части структуры данных, которая содержит такую ссылку.

Система обеспечивает специфический удаленный объект, реестр RMI, для нахождения ссылок на удаленные объекты. Реестр RMI - это простой сервис имен удаленных объектов, который позволяет удаленным клиентам получать ссылку на удаленный объект по имени. Реестр RMI используется обычно, чтобы получить ссылку на первый удаленный объект, который нужно использовать клиенту RMI. Этот первый удаленный объект затем обеспечивает нахождение остальных объектов.

Интерфейсы java.rmi.registry.LocateRegistry и java.rmi.registry.Registry выступают как связующие API для связывания, регистрации, поиска удаленных объектов в реестре. Если удаленный объект зарегистрирован в реестре RMI на локальном хосте, вызывающие объекты на любом хосте могут искать удаленный объект по имени, получать его ссылку и затем вызывать методы объекта. Реестр может совместно использоваться всеми серверами, выполняющимися на хосте, или же каждый сервер может создавать и использовать свой реестр.

```
String name = "Compute";
```

Имя Compute идентифицирует удаленный объект в реестре. Программа затем должна добавить имя в реестр RMI, выполняющийся на сервере. Это делается позднее (в блоке try) в операторе:

```
registry.rebind(name, stub);
```

Вызов метода rebind производит удаленный вызов реестра RMI на локальном хосте. Этот вызов может приводить к генерации RemoteException, так что должно обрабатываться исключение. Класс ComputeEngine обрабатывает исключение в блоке try/catch.

Об аргументах вызова registry.rebind(name, stub);

Первым параметром является java.lang.String в формате URL, представляющая имя удаленного объекта.

Среда выполнения RMI замещает ссылки на заглушки для удаленных объектов, указанных в аргументах. Удаленно реализованные объекты, такие как экземпляры ComputeEngine, никогда не покидают виртуальную машину, в которой они созданы, так что, когда клиент выполняет поиск в реестре удаленных объектов сервера, возвращается ссылка на заглушку. Как обсуждалось раньше, удаленные объекты в таких случаях передаются по ссылке, а не по значению.

Из соображений безопасности приложение может сделать bind, unbind или rebind ссылки удаленных объектов только в реестре, выполняющемся на том же хосте. Это ограничение предупреждает удаление или замещение удаленным клиентом любых элементов в реестре сервера. Однако, lookup может запрашиваться с любого хоста, локального или удаленного.

Если сервер зарегистрировался в локальном реестре RMI, он печатает сообщение, показывающее, что он готов начать обработку вызовов и что метод main завершился. Нет необходимости иметь нить, ожидающую все время жизни сервера. Пока на другой виртуальной машине имеется ссылка на объект ComputeEngine, объект ComputeEngine не будет уничтожен или отправлен в "мусор". Поскольку программа удерживает ссылку на

ComputeEngine в реестре, она достижима для удаленного клиента. Система RMI заботится о сохранении ComputeEngine работоспособным. ComputeEngine может принимать вызовы и не должен быть перезаявлен до тех пор, пока его связка не будет удалена из реестра и пока удаленные клиенты сохраняют удаленную ссылку на объект ComputeEngine.

Последняя часть кода метода ComputeEngine.main занимается обработкой любых исключений, которые могут возникать. Единственное исключение, которое может возникать в коде - это RemoteException, выбрасываемое либо конструктором класса ComputeEngine, либо вызовом реестра RMI для связывания объекта с именем Compute. В обоих случаях программа не может сделать ничего, кроме печати сообщения об ошибке и завершения. В некоторых распределенных приложениях возможно восстановление после ошибки удаленного вызова. Например, приложение может выбрать другой сервер и продолжить работу.

Создание клиентской программы

Вычислитель является прелестным примером программы: он выполняет задачи, которые ему передаются. Клиенты вычислителя более сложны. Клиент должен вызвать вычислитель, но он также должен определить задачу, которая должна быть выполнена вычислителем.

Два отдельные класса составляют клиента в нашем примере. Первый класс, ComputePi, ищет и вызывает объект Compute. Второй класс, Pi, реализует интерфейс Task<T> и определяет работу, которая должна быть сделана вычислителем. Работой класса Pi является вычисление значения π до некоторого десятичного знака.

Как вы помните, неудаленный интерфейс Task<T> определяется как:

```
package compute;

public interface Task<T> {
   T execute();
}
```

Применительно к объекту, ссылка на который размещается в переменной comp типа Compute, вызывается метод, который должен получать ссылку на объект реализующий интерфейс Task<T> (далее этот объект сериализуется и его копия передаётся на хост вычислителя, где по выполнению поставленной задачи вычислителем копия результата решения задачи опять же по средствам сериализации передаётся обратно на хост клиента). Определение задачи Pi, реализующей интерфейс Task<T>, показано ниже. Объект Pi конструируется с единственным аргументом — желаемая точность результата. Результатом выполнения задачи является число типа java.math.BigDecimal, вычисленное с заданной точностью.

Клиентский класс client.ComputePi следующий:

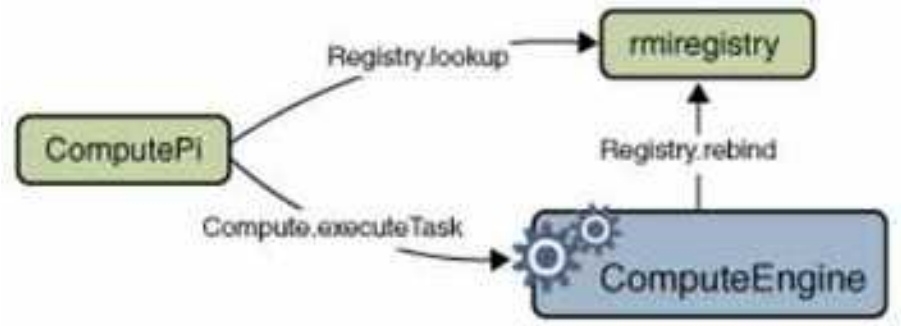
```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;
public class ComputePi {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new SecurityManager());
      }
```

```
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Как и сервер ComputeEngine, клиент начинается с инсталляции менеджера безопасности. Это необходимо потому, что RMI может загрузить код на клиента. В этом примере на клиента загружается заглушка ComputeEngine. Всегда, если любой код загружается через RMI, должен быть представлен менеджер безопасности.

После инсталляции менеджера безопасности клиент конструирует имя, используя поиск удаленного объекта Compute. Значение первого аргумента командной строки, args[0], является именем удаленного хоста, на котором выполняется объект Compute. Клиент использует метод registry.lookup для поиска удаленного объекта по имени в реестре удаленного хоста. Когда выполняется поиск по имени, код создает URL, который определяет хост, на котором выполняется сервер вычислителя. Имя, передаваемое в вызов registry.lookup имеет тот же синтаксис URL, что и имя, передаваемое в вызов registry.rebind, который обсуждался ранее.

Следующим шагом клиент создает объект Pi, передавая в конструктор Pi второй аргумент командной строки, args[1], который показывает число десятичных цифр для вычисления. Наконец, клиент вызывает метод executeTask удаленного объекта Compute. Объект, переданный в вызов executeTask возвращает объект типа java.math.BigDecimal, так что программа записывает возвращенное значение в переменную pi. Затем программа печатает результат. Следующий рисунок показывает поток сообщений между клиентом ComputePi, rmiregistry и ComputeEngine.



Наконец, давайте рассмотрим смысл всего этого: класс Pi. Этот класс реализует интерфейс Task<T> и вычисляет значение до заданной десятичной цифры. Для данного примера действительный алгоритм неважен, не считая, конечно, точности вычисления. Все, что важно, - это то, что процедура довольно дорогостоящая с точки зрения объема вычислений и это одна из тех вещей, из-за которых вы можете захотеть получить более мощный сервер.

Вот код класса client.Pi, который реализует Task<T>.

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {

    private static final long serialVersionUID = 227L;

    /** constants used in pi computation */
    private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

    /** rounding mode to use during pi computation */
    private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

    /** digits of precision after the decimal point */
    private final int digits;

    /**
     * Construct a task to calculate pi to the specified
     * precision.
     */
    public Pi(int digits) {
        this.digits = digits;
    }

    /**
     * Calculate pi.
     */
    public BigDecimal execute() {
        return computePi(digits);
    }

    /**
     * Compute the value of pi to the specified number of
     * digits after the decimal point.  The value is
     * computed using Machin's formula:
     *
     *          pi/4 = 4*arctan(1/5) - arctan(1/239)
     *
     * and a power series expansion of arctan(x) to
     * sufficient precision.
     */
    public static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        BigDecimal arctan1_5 = arctan(5, scale);
        BigDecimal arctan1_239 = arctan(239, scale);
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
```

```
                                arctan1_239).multiply(FOUR);
        return pi.setScale(digits,
                           BigDecimal.ROUND_HALF_UP);
    }
    /**
     * Compute the value, in radians, of the arctangent of
     * the inverse of the supplied integer to the specified
     * number of digits after the decimal point.  The value
     * is computed using the power series expansion for the
     * arc tangent:
     *
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
     *     (x^9)/9 ...
     */
    public static BigDecimal arctan(int inverseX,
                                    int scale)
    {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2 =
            BigDecimal.valueOf(inverseX * inverseX);

        numer = BigDecimal.ONE.divide(invX,
                                      scale, roundingMode);

        result = numer;
        int i = 1;
        do {
            numer =
                numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term =
                numer.divide(BigDecimal.valueOf(denom),
                             scale, roundingMode);
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }
}
```

Наиболее интересное свойство этого примера - в том, что объект Compute никогда не нуждается в определении класса Pi, пока объект Pi не передается как аргумент методу executeTask. В этой точке программы класс загружается RMI в виртуальную машину объекта Compute, вызывается метод execute и код задачи выполняется. Результирующий объект, который в случае задачи Pi является объектом java.math.BigDecimal, передается обратно вызвавшему клиенту, где он используется для печати результата вычислений.

Тот факт, что передаваемый объект Task<T> вычисляет значение Pi не имеет значения для объекта ComputeEngine. Вы можете также реализовать задачу, которая, например, будет генерировать случайное число, используя вероятностный алгоритм. Она будет также работой с интенсивными вычислениями и, следовательно будет, кандидатом на передачу ComputeEngine, но она будет включать в себя совершенно другой код. Этот код также может быть скачан, когда объект Task будет передан объекту Compute. Объект Compute знает только, что каждый объект, который он получает, реализует метод execute; он не знает и ему не нужно знать, что эта реализация делает.

Компиляция программ примера

В реальном сценарии, когда сервис, подобный вычислителю, устанавливается, установщик будет скорее всего создавать JAR-файл, содержащий интерфейсы Compute и Task для классов сервера и клиентскую программу. Далее установщик, возможно, тот же самый создатель JAR-файла, напишет реализацию интерфейса Compute и установит этот сервис на машине, доступной для клиентов. Разработчики клиентских программ смогут использовать интерфейсы Compute и Task, содержащиеся в JAR-файле, и независимо разрабатывать задачи и клиентские программы, которые используют сервис Compute.

Пример разделяет интерфейсы, реализацию удаленного объекта и клиентский код в три пакета:

- compute (интерфейсы Compute и Task)
- engine (класс реализации и заглушка ComputeEngine)
- client (клиентский код ComputePi и реализация задачи Pi)

Давайте сначала соберем JAR-файл интерфейсов, чтобы обеспечить разработчиков сервера и клиентов.

Сборка JAR-файла интерфейсных классов

Прежде всего вам необходимо компилировать исходные файлы интерфейсов в пакете compute, а затем из них собрать JAR-файл. Предположим, каталог с файлами исходного кода следующий:

```
c:\work\compute
```

При данном маршруте вы можете использовать следующие команды для компиляции интерфейсов и создания JAR-файла.

```
cd c:\work
javac compute\*.java
jar cvf compute.jar compute\*.class
```

Теперь вы можете передавать файл compute.jar разработчикам сервера и клиентских приложений, так что они могут использовать эти интерфейсы.

Когда вы собираете либо серверные, либо клиентские классы, вам обычно необходимо указать, где должны располагаться результирующие файлы классов, чтобы они были доступными в сети. В нашем примере они будут располагаться на общем сетевом диске . Файловые URL имеют форму:

file:/P:/ПапкаДляОбмена/home23/

Сборка серверных и клиентских классов

В рамках данной работы не важно каким образом вы получите байт-код из исходных файлов интерфейса, сервера и клиента (консоль, привлечение IDE), главное, чтобы он у вас был. Далее его нужно будет правильно распределить на хосте сервера, клиента и сетевой папки.

Пример. Команды, размещение файлов с байт-кодом (в том числе, содержащимся в jar-файле) и содержимое файлов безопасности

На следующем рисунке схематично показано расположение различных файлов:

Далее приводится последовательность команд для запуска.

Запуск rmiregistry из деректории, где интерфейс "compute" (т.е. запуск cmd и переход в директорию, где находится интерфейс "compute"):

>rmiregistry

Запуск сервера:

1.1) установка переменной окружения CLASSPATH - путь к классам сервера (интерфейс "compute" не в jar-файле):

>set CLASSPATH=d:\Temp\home\server\src;d:\Temp\home\server\public\classes

1.2) непосредственно запуск сервера:

>java
[-cp d:\Temp\home\server\src;d:\Temp\home\server\public\classes]
-Djava.rmi.server.codebase=file:/d:/Temp/home/client/public/classes/
[-Djava.rmi.server.hostname=localhost]
-Djava.security.policy=d:\Temp\home\server\src\server.policy engine.ComputeEngine

Запуск клиента:

2.1) установка переменной окружения CLASSPATH - путь к классам клиента (интерфейс "compute" в jar-файле):

set CLASSPATH=d:\Temp\home\client\src;d:\Temp\home\client\public\classes\compute.jar
2.2) непосредственно запуск клиента:
>java
[-cp d:\Temp\home\client\src;d:\Temp\home\client\public\classes\compute.jar]
[-Djava.rmi.server.codebase=file:/d:/Temp/home/client/public/classes/]
-Djava.security.policy=d:\Temp\home\client\src\client.policy client.ComputePi localhost 45

"C:\Program Files\Java\jdk1.8.0_20\bin\rmiregistry.exe"
"C:\Program Files\Java\jdk1.8.0_20\bin\javac.exe"
"C:\Program Files\Java\jdk1.8.0_20\bin\java.exe"

Содержимое файла server.policy:
grant {permission java.security.AllPermission;};

Содержимое файла client.policy:
grant {permission java.security.AllPermission;};

::
::Пример команд для запуска в задании №64::
::

############rmiregistry
Папка compute лежит в P:\ПапкаДляОбмена\home23\interface\bin переходим в неё в коммандной строке и запускаем rmiregistry:
C:\Work\home23\interface\bin>P:
P:\>cd P:\ПапкаДляОбмена\home23\interface\bin
P:\ПапкаДляОбмена\home23\interface\bin>"C:\Program Files\Java\jdk1.8.0_20\bin\rmiregistry.exe"

############server

set CLASSPATH=C:\Work\home23\server\bin;P:\ПапкаДляОбмена\home23\interface\compute.jar

java -cp C:\Work\home23\server\bin;P:\ПапкаДляОбмена\home23\interface\compute.jar
-Djava.rmi.server.codebase=file:/P:/ПапкаДляОбмена/home23/client/bin/
-Djava.rmi.server.hostname=WS325-06
-Djava.security.policy=C:\Work\home23\server\bin\server.policy engine.ComputeEngine

#############client

set CLASSPATH=C:\Work\home23\client\bin;P:\ПапкаДляОбмена\home23\interface\compute.jar

java -cp C:\Work\home23\client\bin;P:\ПапкаДляОбмена\home23\interface\compute.jar
-Djava.rmi.server.codebase=file:/P:/ПапкаДляОбмена/home23/client/bin/
-Djava.security.policy=P:\ПапкаДляОбмена\home23\client.policy client.ComputePi WS325-06 45

Практическая часть

1. Изучить теоретическую часть. Запустить распределённое приложение из примера из теоретической части, показать преподавателю, запуск осуществить на одном хосте.
2. Задание, полученное в первой лабораторной работе, необходимо представить как задачу «Task», которая должна решаться вычислителем.
3. Распределённое приложение должно запускаться минимум на двух различных компьютерах (клиентов м.б. сколько угодно).
4. При реализации клиента параметры его запуска (адрес сервера и прочее в зависимости от задания) передавать через командную строку.

5. Исходные файлы пакетов, а также файлы байт-кода должны находится в отдельных каталогах. Например: «d:\Temp\home\server\», «d:\Temp\home\client\» и «d:\Temp\home\interface\».
6. Создать либо же нет, в зависимости от варианта, jar-файл пакета «compute»: 0 - не создавать, 1 - создавать. Если требуется создавать jar-файл, то при запуске приложений сервера и клиента у них не должно быть доступа к файлам «Compute.class» и «Task.class», находящимся вне созданного jar-файла (сброс и установка нужного CLASSPATH).
7. Размещение файлов пакета «compute» (либо jar-файл пакета «compute», что зависит от п.6) для клиента: 0 - в локальной директории компьютера клиента, 1 – в каталоге в сетевой папке обмена.
8. Размещение файлов пакета «compute» (либо jar-файл пакета «compute», что зависит от п.6) для сервера: 0 - в локальной директории компьютера сервера, 1 – в каталоге в сетевой папке обмена.
9. Размещение файла безопасности клиента: 0 - в локальной директории клиента, 1 – в каталоге в сетевой папке обмена.
10. Размещение файла безопасности сервера: 0 - в локальной директории сервера, 1 – в каталоге в сетевой папке обмена.
11. Размещение файла байт-кода задачи «Task» (в примере это «client\Pi.class»): 0 - в локальной директории компьютера сервера, 1 – в каталоге в сетевой папке обмена.

Варианты заданий

Таблица №1

№	(П.6)	(П.7)	(П.8)	(П.9)	(П.10)	(П.11)
1.	0	0	0	0	0	0
2.	1	0	0	0	1	1
3.	0	1	0	0	1	0
4.	1	1	0	0	0	1
5.	0	0	1	0	0	0
6.	1	0	1	0	1	0
7.	0	1	1	0	0	1
8.	1	1	1	0	1	0
9.	0	0	0	1	1	1
10.	1	0	0	1	0	0
11.	0	1	0	1	0	0
12.	1	1	0	1	1	1
13.	0	0	1	1	0	0
14.	1	0	1	1	1	1
15.	0	1	1	1	1	0
16.	1	1	1	1	0	0
17.	0	0	0	0	0	1
18.	1	0	0	0	1	0
19.	0	1	0	0	0	1
20.	1	1	0	0	1	0
21.	0	0	1	0	1	0
22.	1	0	1	0	0	1
23.	0	1	1	0	0	0
24.	1	1	1	0	1	1
25.	0	0	0	1	0	0
26.	1	0	0	1	1	0
27.	0	1	0	1	1	1
28.	1	1	0	1	0	0

29.	0	0	1	1	0	1
30.	1	0	1	1	1	0
31.	0	1	1	1	0	0
32.	1	1	1	1	1	1
33.	0	0	0	0	1	0
34.	1	0	0	0	0	1
35.	0	1	0	0	0	0
36.	1	1	0	0	1	0
37.	0	0	1	0	0	1
38.	1	0	1	0	1	0
39.	0	1	1	0	1	1
40.	1	1	1	0	0	0
41.	0	0	0	1	0	0
42.	1	0	0	1	1	1
43.	0	1	0	1	0	0
44.	1	1	0	1	1	1
45.	0	0	1	1	1	0
46.	1	0	1	1	0	0
47.	0	1	1	1	0	1
48.	1	1	1	1	1	0
49.	0	0	0	0	0	1
50.	1	0	0	0	1	0
51.	0	1	0	0	1	0
52.	1	1	0	0	0	1
53.	0	0	1	0	0	0
54.	1	1	1	0	1	1
55.	0	1	1	0	0	0
56.	1	1	1	0	1	0
57.	0	1	0	1	1	1
58.	1	1	0	1	0	0
59.	0	1	0	1	0	1
60.	1	1	0	1	1	0
61.	0	1	1	1	0	0
62.	1	1	1	1	1	1
63.	0	1	1	1	1	0
64.	1	1	1	1	0	1